

# Efficient Computation of Moments in Sum-Product Networks

Han Zhao  
Geoff Gordon

HAN.ZHAO@CS.CMU.EDU  
GGORDON@CS.CMU.EDU

*Machine Learning Department, Carnegie Mellon University, Pittsburgh, PA, USA*

## Abstract

Bayesian online learning algorithms for Sum-Product Networks (SPNs) need to compute moments of model parameters under the one-step update posterior distribution. The best existing method for computing such moments scales quadratically in the size of the SPN, although it scales linearly for trees. We propose a linear-time algorithm that works even when the SPN is a directed acyclic graph (DAG). We achieve this goal by reducing the moment computation problem into a joint inference problem in SPNs and by taking advantage of a special structure of the one-step update posterior distribution: it is a multilinear polynomial with exponentially many monomials, and we can evaluate moments by differentiating. The latter is known as the *differential trick*. We apply the proposed algorithm to develop a linear time assumed density filter (ADF) for SPN parameter learning. As an additional contribution, we conduct extensive experiments comparing seven different online learning algorithms for SPNs on 20 benchmark datasets. The new linear-time ADF method consistently achieves low runtime due to the efficient linear-time algorithm for moment computation; however, we discover that two other methods (CCCP and SMA) typically perform better statistically, while a third (BMM) is comparable to ADF. Interestingly, CCCP can be viewed as implicitly using the same differentiation trick that we make explicit here. The fact that two of the top four fastest methods use this trick suggests that the same trick might find other uses for SPN learning in the future.

## 1. Introduction

Sum-Product Networks (SPNs) have recently attracted some interest because of their flexibility in modeling complex distributions as well as the tractability of performing exact marginal inference (Poon and Domingos, 2011; Gens and Domingos, 2012, 2013; Peharz et al., 2015; Zhao et al., 2015, 2016a,b; Peharz et al., 2016). They are general-purpose inference machines over which one can perform exact joint, marginal and conditional queries in linear time in the size of the network. It has been shown that discrete SPNs are equivalent to arithmetic circuits (ACs) (Darwiche, 2003; Park and Darwiche, 2004) in the sense that one can transform each SPN into an equivalent AC and vice versa in linear time and space with respect to the network size (Rooshenas and Lowd, 2014). SPNs are also closely connected to probabilistic graphical models: by interpreting each sum node in the network as a hidden variable and each product node as a rule encoding context-specific conditional independence (Boutilier et al., 1996), every SPN can be equivalently converted into a Bayesian network where compact data structures are used to represent the local probability distributions (Zhao et al., 2015). This relationship characterizes the probabilistic semantics encoded by the network structure and allows practitioners to design principled and efficient parameter learning algorithms for SPNs (Zhao et al., 2016a,b).

Most of the existing batch learning algorithms can be straightforwardly adapted to the online setting where the network updates its parameters after it receives one instance at each time step. This online learning setting makes SPNs more widely applicable in various real-world scenarios. This includes

the case where either the data set is too large to store at once, or the network needs to adapt to the change of external data distributions. Recently Rashwan et al. (2016) proposed an online Bayesian Moment Matching (BMM) algorithm to learn the probability distribution of the model parameters of SPNs based on the method of moments. Later Jaini et al. (2016) extended this algorithm to the continuous case where the leaf nodes in the network are assumed to be Gaussian distributions. Empirically they show that BMM is superior to online extensions of projected gradient descent and exponentiated gradient. At a high level BMM can be understood as an instance of the general assumed density filtering framework (Sorenson and Stubberud, 1968) where the algorithm finds an approximate posterior distribution within a tractable family of distributions by the method of moments. Specifically, BMM for SPNs works by matching the first and second order moments of the approximate tractable posterior distribution to the exact but intractable posterior.

An essential sub-routine of the above two algorithms (Rashwan et al., 2016; Jaini et al., 2016) is to efficiently compute the exact first and second order moments of the one-step update posterior distribution (cf. 3.2). Rashwan et al. (2016) designed a recursive algorithm to achieve this goal in linear time when the underlying network structure is a tree, and this algorithm is also used by Jaini et al. (2016) in the continuous case. However, the algorithm only works when the underlying network structure is a tree, and a naive computation of such moments in a DAG will scale quadratically w.r.t. the network size. Often this quadratic computation is prohibitively expensive even for SPNs with moderate sizes.

In this paper we propose a linear time and space algorithm that is able to compute, under mild assumptions, any moments of all the network parameters simultaneously even when the underlying network structure is a DAG. The key technique in the design of our algorithm dates back to the differential approach (Darwiche, 2003) used for exact inference in Bayesian networks. This technique has also been implicitly used in the recent development of a concave-convex procedure (CCCP) for optimizing the weights of SPNs (Zhao et al., 2016b). Essentially, by reducing the moment computation problem into a joint inference problem in SPNs, we are able to exploit the fact that the network polynomial of an SPN is a multilinear function in terms of model parameters, so we can efficiently evaluate this polynomial by differentiation even if the polynomial may contain exponentially many monomials, provided that the polynomial admits a tractable circuit complexity. To illustrate this, let us start with a simple example. A formal proof will be given in Sec. 3.3.

**Example 1.1.** Assume we have a multilinear function  $g$  of  $(w_1, w_2, w_3, w_4)$  as  $g(w_1, w_2, w_3, w_4) = w_1(3w_2 + 4w_3) + w_4(5w_2 + 6w_3)$ . We want to select all the monomials inside  $g$  that contain  $w_1$ . A brute force approach is to expand  $g$ , and check sequentially each monomial to see whether it contains  $w_1$  or not, and then remove those monomials that do not contain  $w_1$ . But we can do better than brute force, by using the following differential trick. First, compute  $\frac{\partial g}{\partial w_1} = 3w_2 + 4w_3$ , and then multiply back  $w_1$  into  $\frac{\partial g}{\partial w_1}$ , which gives us:  $w_1 \frac{\partial g}{\partial w_1} = w_1(3w_2 + 4w_3)$ . As long as  $g$  admits a tractable circuit complexity, i.e.,  $g$  has a circuit representation whose size is much smaller than the total number of monomials in the polynomial, we can skip all the factors that do not contain  $w_1$  when doing the differentiation, and this will save us unnecessary computations that do not contribute to the final answer.

As we will see shortly, the network polynomial function computed by an SPN is indeed a multilinear function in terms of both model weights and the leaf indicators. Furthermore, the network structure gives us a natural and tractable factorization of this network polynomial, allowing us to be able to apply the above differential trick efficiently.

With the linear time sub-routine for computing moments, we are able to design an assumed density filter (Sorenson and Stubberud, 1968) (ADF) to learn the parameters of SPNs in an online fashion. ADF runs in linear time and space due to our efficient sub-routine. We also show that ADF and BMM can both be understood under a general framework of moment matching, where the only difference lies in the moments chosen to be matched and how to match the chosen moments. Finally, we make an extensive empirical comparison among all the online learning algorithms proposed to date for SPNs on 20 benchmark data sets in terms of both computational efficiency as well as statistical accuracy. We show that by using the proposed sub-routine for moment computations, ADF and BMM run faster than all the competitors as the data gets larger and larger. This extensive empirical comparison will also help practitioners to choose the most suitable algorithm when faced with various application scenarios.

## 2. Preliminaries

We use  $[n]$  to abbreviate  $\{1, 2, \dots, n\}$ . We use a capital letter  $X$  to denote a random variable and a bold capital letter  $\mathbf{X}$  to denote a set of random variables (random vector). Similarly, a lowercase letter  $x$  is used to denote a value taken by  $X$  and a bold lowercase letter  $\mathbf{x}$  denotes a joint value taken by the corresponding random vector  $\mathbf{X}$ . We use calligraphic letters to denote graphs (e.g.,  $\mathcal{G}$ ). In particular, we reserve  $\mathcal{S}$  to represent an SPN, and we use  $h$  to mean the height of  $\mathcal{S}$ .  $|\mathcal{S}|$  is the size of an SPN, i.e., the number of edges plus the number of nodes in the graph.

### 2.1 Sum-Product Networks

A sum-product network  $\mathcal{S}$  is a computational circuit over a set of random variables  $\mathbf{X} = \{X_1, \dots, X_n\}$ . It is a rooted directed acyclic graph. The internal nodes of  $\mathcal{S}$  are sums or products and the leaves are univariate distributions over  $X_i$ . In its simplest form, the leaves of  $\mathcal{S}$  are indicator variables  $\mathbb{I}_{X=x}$ , which can also be understood as categorical distributions whose entire probability mass is on a single value. Edges from sum nodes are parameterized with positive weights. Let  $\mathbf{x}$  be an instantiation of the random vector  $\mathbf{X}$ . We associate an unnormalized probability, or density,  $V_k(\mathbf{x} \mid \mathbf{w})$  with each node  $k$  when the input to the network is  $\mathbf{x}$  with network weights set to be  $\mathbf{w}$ :

$$V_k(\mathbf{x} \mid \mathbf{w}) = \begin{cases} p(X_i = \mathbf{x}_i) & k \text{ is a leaf node over } X_i \\ \prod_{j \in \text{Ch}(k)} V_j(\mathbf{x} \mid \mathbf{w}) & k \text{ is a product node} \\ \sum_{j \in \text{Ch}(k)} w_{kj} V_j(\mathbf{x} \mid \mathbf{w}) & k \text{ is a sum node} \end{cases} \quad (1)$$

where  $\text{Ch}(k)$  is the child list of node  $k$  in the graph and  $w_{kj}$  is the edge weight associated with sum node  $k$  and its child node  $j$ . The probability/density of a joint assignment  $\mathbf{X} = \mathbf{x}$  is computed by the value at the root of  $\mathcal{S}$  with input  $\mathbf{x}$  divided by a normalization constant  $V_{\text{root}}(\mathbf{1} \mid \mathbf{w})$ :

$$p(\mathbf{x}) = \frac{V_{\text{root}}(\mathbf{x} \mid \mathbf{w})}{V_{\text{root}}(\mathbf{1} \mid \mathbf{w})} \quad (2)$$

where  $V_{\text{root}}(\mathbf{1} \mid \mathbf{w})$  is the value of the root node when all the probabilities/densities at the leaf nodes are set to be 1. This essentially corresponds to marginalizing/integrating out the random vector  $\mathbf{X}$ , which will ensure Eq. 2 defines a proper probability distribution. Eq. 2 can also be used to compute the marginal probability of a partial assignment  $\mathbf{Y} = \mathbf{y}$ : simply set  $V$  at leaf nodes whose corresponding random variable is not in  $\mathbf{Y}$  to be 1 and other leaf nodes based on the assignment

$\mathbf{Y} = \mathbf{y}$ . Again, this corresponds to integrating out variables outside of the partial assignment. As a corollary, conditional probabilities can also be computed by evaluating two partial assignments. Due to this property, joint, marginal, and conditional queries can all be answered in linear time in the size of the network.

## 2.2 Bayesian Networks and Mixture Models

We provide two alternative interpretations of SPNs that will be useful later to design our linear time moment computation algorithm. The first one relates SPNs with Bayesian networks (BNs). Informally, any complete and decomposable SPN  $\mathcal{S}$  over  $\mathbf{X} = \{X_1, \dots, X_n\}$  can be converted into a bipartite BN with  $O(n|\mathcal{S}|)$  size (Zhao et al., 2015). Each internal sum node in  $\mathcal{S}$  corresponds to one latent variable in the constructed BN, and each leaf distribution node corresponds to one observable variable in the BN. Furthermore, the constructed BN will be a simple bipartite graph with one layer of local latent variables pointing to one layer of observable variables  $\mathbf{X}$ . An observable variable is a child of a local latent variable if and only if the observable variable appears as a descendant of the latent variable (sum node) in the original SPN. This means that the SPN  $\mathcal{S}$  can be understood as a BN where the number of latent variables per instance is  $O(|\mathcal{S}|)$ . Since the constructed BN is a directed bipartite graph, we know that all the local latent variables are independent from each other given that no observation is made. Or equivalently, taking a Bayesian perspective, the prior distribution over the model parameters should be fully decomposable if no observation on  $\mathbf{X}$  is known.

The second perspective is to view an SPN  $\mathcal{S}$  as a mixture model with exponentially many mixture components (Dennis and Ventura, 2015; Zhao et al., 2016b). More specifically, we can decompose each complete and decomposable SPN  $\mathcal{S}$  into a sum of induced trees, where each tree corresponds to a product of univariate distributions. To proceed, we first formally define what we called *induced trees*, which will also be useful in our later discussion and derivation.

**Definition 2.1** (Induced tree SPN (Zhao et al., 2016a)). Given a complete and decomposable SPN  $\mathcal{S}$  over  $\mathbf{X} = \{X_1, \dots, X_n\}$ ,  $\mathcal{T} = (\mathcal{T}_V, \mathcal{T}_E)$  is called an *induced tree SPN* from  $\mathcal{S}$  if

1.  $\text{Root}(\mathcal{S}) \in \mathcal{T}_V$ .
2. If  $v \in \mathcal{T}_V$  is a sum node, then exactly one child of  $v$  in  $\mathcal{S}$  is in  $\mathcal{T}_V$ , and the corresponding edge is in  $\mathcal{T}_E$ .
3. If  $v \in \mathcal{T}_V$  is a product node, then all the children of  $v$  in  $\mathcal{S}$  are in  $\mathcal{T}_V$ , and the corresponding edges are in  $\mathcal{T}_E$ .

Here  $\mathcal{T}_V$  is the node set of  $\mathcal{T}$  and  $\mathcal{T}_E$  is the edge set of  $\mathcal{T}$ .

It has been shown that Def. 2.1 produces subgraphs of  $\mathcal{S}$  that are trees as long as the original SPN  $\mathcal{S}$  is complete and decomposable. One useful result based on the concept of induced trees is:

**Theorem 2.1** ((Zhao et al., 2016b)). Let  $\tau_{\mathcal{S}} = V_{\text{root}}(\mathbf{1} \mid \mathbf{1})$ . Then  $V_{\text{root}}(\mathbf{x} \mid \mathbf{w})$  can be written as  $\sum_{t=1}^{\tau_{\mathcal{S}}} \prod_{(k,j) \in \mathcal{T}_{tE}} w_{kj} \prod_{i=1}^n p_t(X_i = \mathbf{x}_i)$ , where  $\mathcal{T}_t$  is the  $t$ th unique induced tree of  $\mathcal{S}$  and  $p_t(X_i)$  is a univariate distribution over  $X_i$  in  $\mathcal{T}_t$  as a leaf node.

In Thm. 2.1,  $\tau_{\mathcal{S}}$  equals the number of induced trees in  $\mathcal{S}$ . Without loss of generality assuming that sum layers alternate with product layers in  $\mathcal{S}$ , we have  $\tau_{\mathcal{S}} = \Omega(2^h)$ , where  $h$  is the height of  $\mathcal{S}$ .

Hence the mixture model represented by  $\mathcal{S}$  has number of mixture components that is exponential in the height of  $\mathcal{S}$ . Thm. 2.1 characterizes both the number of components and the form of each component in the mixture model, as well as their mixture weights. For the convenience of later discussion, we call  $V_{\text{root}}(\mathbf{x} \mid \mathbf{w})$  the *network polynomial* of  $\mathcal{S}$ .

**Corollary 2.1.** The *network polynomial*  $V_{\text{root}}(\mathbf{x} \mid \mathbf{w})$  is a multilinear function of  $\mathbf{w}$  with positive coefficients on each monomial.

Corollary 2.1 holds since each monomial corresponds to an induced tree and each edge only appears at most once in the tree. Such a polynomial function is also a special case of a posynomial where the exponents of all the variables are at most 1. This property will be crucial and useful in our derivation of a linear time algorithm for moment computation in SPNs.

### 2.3 Differential Approach for Inference in Bayesian Networks

We briefly introduce the idea of performing inference in BNs using the differential trick we mentioned above. This novel idea is due to Darwiche (2003). We also refer readers to Darwiche (2003) for a more complete and detailed discussion of this approach. We start from a simple BN with 2 binary random variables. Using indicator variable notations, we can represent the joint probability distribution of an BN as the following polynomial function:

$$g(X_1, X_2) = p_{00}\mathbb{I}_{\bar{x}_1}\mathbb{I}_{\bar{x}_2} + p_{01}\mathbb{I}_{\bar{x}_1}\mathbb{I}_{x_2} + p_{10}\mathbb{I}_{x_1}\mathbb{I}_{\bar{x}_2} + p_{11}\mathbb{I}_{x_1}\mathbb{I}_{x_2}$$

where  $p_{ij} = \Pr(X_1 = i, X_2 = j)$ ,  $\forall i, j \in \{0, 1\}$ . Given the joint probability distribution, the above function  $g$  can be viewed as a polynomial in terms of the boolean indicator variables over  $X_1$  and  $X_2$ . Several properties can be observed from such polynomials: 1).  $g(\mathbf{X}) = \Pr(\mathbf{X})$ . 2). For an BN with  $n$  random variables, there are  $2^n$  monomials in  $g$ . 3).  $g(\mathbf{X})$  is a multilinear function in terms of the boolean indicators. 4).  $g(\mathbf{X})$  is a homogeneous polynomial, that is, each monomial has the same degree, which is  $n$  for an BN with  $n$  random variables.

Let consider the probabilistic semantics of partial differentiation of  $g$  w.r.t. a specific indicator, say,  $\mathbb{I}_{x_1}$ :

$$\frac{\partial g}{\partial \mathbb{I}_{x_1}}(X_1, X_2) = p_{10}\mathbb{I}_{\bar{x}_2} + p_{11}\mathbb{I}_{x_2} = \Pr(X_1 = 1, X_2)$$

i.e., the partial differentiation of the network polynomial w.r.t.  $\mathbb{I}_{x_1}$  corresponds to evaluating the joint probability by fixing  $X_1 = 1$ . Note that the above equality holds due to property 1) and 3) listed above. In general, in a BN  $\mathcal{B}$  with joint distribution  $\Pr(\mathbf{X})$ , we have the following theorem hold:

**Theorem 2.2.** (Darwiche, 2003) Let  $\mathcal{B}$  be a Bayesian network representing probability distribution  $\Pr(\mathbf{X})$  and having polynomial  $g$ . For every random variable  $X$  and instantiation  $\mathbf{x}$  of  $\mathbf{X}$ , we have:

$$\frac{\partial g}{\partial \mathbb{I}_x}(\mathbf{x}) = \Pr(X = x, \mathbf{x} - X)$$

where  $\mathbf{x} - X$  means the restriction of  $\mathbf{x}$  on  $\mathbf{X} \setminus \{X\}$ .

In words, if we differentiate the polynomial  $g$  w.r.t. indicator  $\mathbb{I}_x$  and evaluate the result at evidence  $\mathbf{x}$ , we obtain the probability of instantiation  $x, \mathbf{x} - X$ . The above theorem can be extended to second-order derivative, or any higher order derivative, as well:

**Corollary 2.2.** Let  $\mathcal{B}$  be a Bayesian network representing probability distribution  $\Pr(\mathbf{X})$  and having polynomial  $g$ . For every pair of random variable  $X_i \neq X_j$  and instantiation  $\mathbf{x}$  of  $\mathbf{X}$ , we have:

$$\frac{\partial^2 g}{\partial \mathbb{I}_{x_i} \partial \mathbb{I}_{x_j}}(\mathbf{x}) = \Pr(X_i = x_i, X_j = x_j, \mathbf{x} - X_i - X_j)$$

Marginal probability and conditional probability can also be computed via partial differentiation:

**Corollary 2.3.** For every random variable  $X$  and instantiation  $\mathbf{x}$  of  $\mathbf{X}$ , we have:

$$\sum_x \frac{\partial g}{\partial \mathbb{I}_x}(\mathbf{x}) = \Pr(\mathbf{x} - X)$$

$$\frac{\frac{\partial g}{\partial \mathbb{I}_{x'}}(\mathbf{x})}{\sum_x \frac{\partial g}{\partial \mathbb{I}_x}(\mathbf{x})} = \Pr(x' \mid \mathbf{x} - X)$$

The above theorem and corollaries show that we can compute specific forms of joint, marginal and conditional probabilities via partial differentiation w.r.t. specific indicators. However, due to property 2), one may ask what advantages does this alternative computation provide us since there are exponentially many monomials in  $g$  and the cost of a naive differentiation scales linearly in the number of monomials? The insight here is that for many distributions of practical interests we can often find arithmetic circuits/sum-product networks whose size are only polynomial in  $n$ . Formally, the *circuit complexity* (Darwiche, 2003) of a BN  $\mathcal{B}$  is the size of the smallest arithmetic circuit that computes the network polynomial of  $\mathcal{B}$ . Hence for distributions with tractable circuit complexity, we can compute probabilistic queries efficiently as long as we can perform partial differentiation on the circuit efficiently. As we will see shortly, this is the case for both arithmetic circuits and sum-product networks.

### 3. Linear Time Exact Moment Computation

#### 3.1 Exact Posterior Has Exponentially Many Modes

Let  $M$  be the number of sum nodes in  $\mathcal{S}$ . Suppose we are given a fully factorized prior distribution  $p_0(\mathbf{w}; \boldsymbol{\alpha}) = \prod_{k=1}^M p_0(w_k; \alpha_k)$  over  $\mathbf{w}$ . It is worth pointing out the fully factorized prior distribution is well justified by the bipartite graph structure of the equivalent BN we introduced in section 2.2. We are interested in computing the moments of the posterior distribution after we receive one observation from the world. Essentially, this is the Bayesian online learning setting where we update the belief about the distribution of model parameters as we observe data from the world sequentially. Note that  $w_k$  corresponds to the weight vector associated with sum node  $k$ , so  $w_k$  is a vector that satisfies  $w_k > 0$  and  $\mathbf{1}^T w_k = 1$ . For the ease of discussion let us assume that the prior distribution for each  $w_k$  is Dirichlet, i.e.,

$$p_0(\mathbf{w}; \boldsymbol{\alpha}) = \prod_{k=1}^M \text{Dir}(w_k; \alpha_k) = \prod_{k=1}^M \frac{\Gamma(\sum_j \alpha_{k,j})}{\prod_j \Gamma(\alpha_{k,j})} \prod_j w_{k,j}^{\alpha_{k,j}-1}$$

After observing one instance  $\mathbf{x}$ , we have the exact posterior distribution to be:

$$p(\mathbf{w} \mid \mathbf{x}) = \frac{p_0(\mathbf{w}; \boldsymbol{\alpha}) p(\mathbf{x} \mid \mathbf{w})}{p(\mathbf{x})} \quad (3)$$

Let  $Z_{\mathbf{x}} \triangleq p(\mathbf{x})$  and realize that the network polynomial function also computes the likelihood  $p(\mathbf{x} \mid \mathbf{w})$ . Plugging the expression for the prior distribution as well as the network polynomial into the above Bayes formula, we have

$$p(\mathbf{w} \mid \mathbf{x}) = \frac{1}{Z_{\mathbf{x}}} \sum_{t=1}^{\tau_S} \prod_{k=1}^M \text{Dir}(w_k; \alpha_k) \prod_{(k,j) \in \mathcal{T}_{tE}} w_{k,j} \prod_{i=1}^n p_t(x_i)$$

Since Dirichlet is a conjugate distribution to the multinomial, each term in the summation is an updated Dirichlet with a multiplicative constant. So, the above equation suggests that the exact posterior distribution becomes a mixture of  $\tau_S$  Dirichlets after one observation. In a data set of  $D$  instances, the exact posterior will become a mixture of  $\tau_S^D$  components, which is intractable to maintain in practice since  $\tau_S = \Omega(2^h)$ .

The hardness of maintaining the exact posterior distribution appeals for an approximate scheme where we can sequentially update our belief about the distribution while at the same time efficiently maintain the approximation. Assumed density filtering (Sorenson and Stubberud, 1968) is such a framework: the algorithm chooses an approximate distribution from a tractable family of distributions after observing each instance. A typical choice is to match the moments of an approximation to the exact posterior.

### 3.2 The Hardness of Computing Moments

In order to find an approximate distribution to match the moments of the exact posterior, we need to be able to compute those moments under the exact posterior. This is not a problem for traditional mixture models including mixture of Gaussians, latent Dirichlet allocation, etc., since the number of mixture components in those models are assumed to be small. However, this is not the case for SPNs, where the effective number of mixture components is  $\tau_S = \Omega(2^h)$ . In this section we will show how to use the network polynomial of  $\mathcal{S}$  to reduce this complexity to  $O(|\mathcal{S}|)$ .

To simplify the notation, for each  $t \in [\tau_S]$ , we define  $c_t \triangleq \prod_{i=1}^n p_t(x_i)^1$  and

$$u_t \triangleq \int_{\mathbf{w}} p_0(\mathbf{w}) \prod_{(k,j) \in \mathcal{T}_{tE}} w_{k,j} d\mathbf{w}$$

That is,  $c_t$  corresponds to the product of leaf distributions in the  $t$ th induced tree  $\mathcal{T}_t$ , and  $u_t$  is the moment of  $\prod_{(k,j) \in \mathcal{T}_{tE}} w_{k,j}$ , i.e., the product of tree edges, under the prior distribution  $p_0(\mathbf{w})$ . Realizing that the posterior distribution needs to satisfy the normalization constraint, we have:

$$\sum_{t=1}^{\tau_S} c_t \int_{\mathbf{w}} p_0(\mathbf{w}) \prod_{(k,j) \in \mathcal{T}_{tE}} w_{k,j} d\mathbf{w} = \sum_{t=1}^{\tau_S} c_t u_t = Z_{\mathbf{x}} \quad (4)$$

Note that the prior distribution for a sum node is a Dirichlet distribution. In this case we can compute a closed form expression for  $u_t$  as:

$$u_t = \prod_{(k,j) \in \mathcal{T}_{tE}} \frac{\alpha_{k,j}}{\sum_{j'} \alpha_{k,j'}} \quad (5)$$

---

1. We omit the explicit dependency of  $c_t$  on the instance  $\mathbf{x}$ .

More generally, let  $f(\cdot)$  be a function applied to each edge weight in an SPN. We use the notation  $M_p(f)$  to mean the moment of function  $f$  evaluated under distribution  $p$ . For example, if we let  $f(w_{k,j}) = w_{k,j}$ , then  $M_p(f)$  corresponds to the first order moment; or if we let  $f(w_{k,j}) = \log w_{k,j}$ , then  $M_p(f)$  corresponds to the log moment, etc.

We are interested in computing  $M_p(f)$  where  $p = p(\mathbf{w} \mid \mathbf{x})$ , which we call the *one-step update posterior distribution*. More specifically, for each edge weight  $w_{k,j}$ , we would like to compute the following quantity:

$$M_p(f(w_{k,j})) = \int_{\mathbf{w}} f(w_{k,j}) p(\mathbf{w} \mid \mathbf{x}) d\mathbf{w} = \frac{1}{Z_{\mathbf{x}}} \sum_{t=1}^{\tau_S} c_t \int_{\mathbf{w}} p_0(\mathbf{w}) f(w_{k,j}) \prod_{(k',j') \in \mathcal{T}_{tE}} w_{k',j'} d\mathbf{w} \quad (6)$$

We first note that Eq. 6 is not trivial to compute as it involves  $\tau_S = \Omega(2^h)$  terms. Furthermore, in order to conduct moment matching, we need to compute the above moment for each edge  $(k, j)$  from a sum node. A naive computation will lead to a total time complexity  $\Omega(|\mathcal{S}|2^h)$ . A linear time algorithm that uses dynamic programming to compute these moments has been designed by Rashwan et al. (2016) when the underlying structure of  $\mathcal{S}$  is a tree. The algorithm recursively computes the above moments in a top-down fashion along the tree. However, this algorithm breaks down when the graph is a DAG.

### 3.3 Efficient Polynomial Evaluation by Differentiation

In what follows we will present a  $O(|\mathcal{S}|)$  time and space algorithm that is able to compute all the moments simultaneously for general SPNs with DAG structures. We will first show that the moment in Eq. 6 can be expressed as a convex combination of two easily computed moments, then we proceed to use the differential trick to efficiently compute those convex coefficients.

Let us first compute Eq. 6 for a fixed edge  $(k, j)$ . Our strategy is to partition all the induced trees based on whether they contain the tree edge  $(k, j)$  or not. Define  $\mathcal{T}_F = \{\mathcal{T}_t \mid (k, j) \notin \mathcal{T}_t, t \in [\tau_S]\}$  and  $\mathcal{T}_T = \{\mathcal{T}_t \mid (k, j) \in \mathcal{T}_t, t \in [\tau_S]\}$ . In other words,  $\mathcal{T}_F$  corresponds to the set of trees that do not contain edge  $(k, j)$  and  $\mathcal{T}_T$  corresponds to the set of trees that contain edge  $(k, j)$ . Then,

$$\begin{aligned} (6) &= \frac{1}{Z_{\mathbf{x}}} \sum_{\mathcal{T}_t \in \mathcal{T}_T} c_t \int_{\mathbf{w}} p_0(\mathbf{w}) f(w_{k,j}) \prod_{(k',j') \in \mathcal{T}_{tE}} w_{k',j'} d\mathbf{w} \\ &\quad + \frac{1}{Z_{\mathbf{x}}} \sum_{\mathcal{T}_t \in \mathcal{T}_F} c_t \int_{\mathbf{w}} p_0(\mathbf{w}) f(w_{k,j}) \prod_{(k',j') \in \mathcal{T}_{tE}} w_{k',j'} d\mathbf{w} \end{aligned}$$

For the induced trees that contain edge  $(k, j)$ , we have

$$\begin{aligned} &\frac{1}{Z_{\mathbf{x}}} \sum_{\mathcal{T}_t \in \mathcal{T}_T} c_t \int_{\mathbf{w}} p_0(\mathbf{w}) f(w_{k,j}) \prod_{(k',j') \in \mathcal{T}_{tE}} w_{k',j'} d\mathbf{w} \\ &= \frac{1}{Z_{\mathbf{x}}} \sum_{\mathcal{T}_t \in \mathcal{T}_T} c_t \int_{\mathbf{w}_{-k}} p_{-0,k}(\mathbf{w}_{-k}) \prod_{\substack{(k',j') \in \mathcal{T}_{tE} \\ (k',j') \neq (k,j)}} w_{k',j'} d\mathbf{w}_{-k} \times \int_{w_k} p_{0,k}(w_k) w_{k,j} f(w_{k,j}) dw_k \\ &= \frac{1}{Z_{\mathbf{x}}} \sum_{\mathcal{T}_t \in \mathcal{T}_T} c_t u_t M_{p'_{0,k}}(f(w_{k,j})) \end{aligned}$$



where  $p'_{0,k}$  is the one-step update posterior Dirichlet distribution for sum node  $k$  after absorbing the term  $w_{k,j}$ , and  $\mathbf{w}_{-k} = \mathbf{w} \setminus \{w_k\}$ . Similarly, for the induced trees that do not contain the edge  $(k, j)$ , we have:

$$\begin{aligned} & \frac{1}{Z_{\mathbf{x}}} \sum_{\mathcal{T}_t \in \mathcal{T}_F} c_t \int_{\mathbf{w}} p_0(\mathbf{w}) f(w_{k,j}) \prod_{(k',j') \in \mathcal{T}_{tE}} w_{k',j'} d\mathbf{w} \\ &= \frac{1}{Z_{\mathbf{x}}} \sum_{\mathcal{T}_t \in \mathcal{T}_F} c_t \int_{\mathbf{w}_{-k}} p_{-0,k}(\mathbf{w}_{-k}) \prod_{(k',j') \in \mathcal{T}_{tE}} w_{k',j'} d\mathbf{w}_{-k} \times \int_{w_k} p_{0,k}(w_k) f(w_{k,j}) dw_k \\ &= \frac{1}{Z_{\mathbf{x}}} \sum_{\mathcal{T}_t \in \mathcal{T}_F} c_t u_t M_{p_{0,k}}(f(w_{k,j})) \end{aligned}$$

where  $p_{-0,k} = p_0/p_{0,k}$  and  $p_{0,k}$  is the prior Dirichlet distribution for sum node  $k$ . The second equation holds by changing the order of integration. The third equation holds because  $(k, j)$  is not in tree  $\mathcal{T}_t$  so that  $\prod_{(k',j') \in \mathcal{T}_{tE}} w_{k',j'}$  does not contain the term  $w_{k,j}$ . Note that both  $M_{p_{0,k}}(f)$  and  $M_{p'_{0,k}}(f)$  are independent of specific induced trees, so we can combine the above two parts to express  $M_p(f)$  as:

$$M_p(f) = \left( \frac{1}{Z_{\mathbf{x}}} \sum_{\mathcal{T}_t \in \mathcal{T}_F} c_t u_t \right) M_{p_{0,k}}(f) + \left( \frac{1}{Z_{\mathbf{x}}} \sum_{\mathcal{T}_t \in \mathcal{T}_T} c_t u_t \right) M_{p'_{0,k}}(f) \quad (7)$$

From Eq. 4 we have

$$\frac{1}{Z_{\mathbf{x}}} \sum_{t=1}^{\tau_S} c_t u_t = 1 \quad \text{and} \quad \sum_{t=1}^{\tau_S} c_t u_t = \sum_{\mathcal{T}_t \in \mathcal{T}_T} c_t u_t + \sum_{\mathcal{T}_t \in \mathcal{T}_F} c_t u_t$$

This implies that  $M_p(f)$  is in fact a convex combination of  $M_{p_{0,k}}(f)$  and  $M_{p'_{0,k}}(f)$ . So in order to compute Eq. 6, we need to be able to compute the two coefficients efficiently. Recall that for each induced tree  $\mathcal{T}_t$ , we have the expression of  $u_t$  as

$$u_t = \prod_{(k,j) \in \mathcal{T}_{tE}} \frac{\alpha_{k,j}}{\sum_{j'} \alpha_{k,j'}}$$

The term  $\sum_{t=1}^{\tau_S} c_t u_t$  can be expressed as

$$\sum_{t=1}^{\tau_S} c_t u_t = \sum_{t=1}^{\tau_S} \prod_{(k,j) \in \mathcal{T}_{tE}} \frac{\alpha_{k,j}}{\sum_{j'} \alpha_{k,j'}} \prod_{i=1}^n p_t(x_i) \quad (8)$$

**Lemma 3.1.**  $\sum_{t=1}^{\tau_S} c_t u_t$  can be computed in  $O(|\mathcal{S}|)$  time and space in a bottom-up evaluation pass of  $\mathcal{S}$ .

*Proof.* Compare the form of Eq. 8 to the network polynomial:

$$p(\mathbf{x}|\mathbf{w}) = V_{\text{root}}(\mathbf{x} | \mathbf{w}) = \sum_{t=1}^{\tau_S} \prod_{(k,j) \in \mathcal{T}_{tE}} w_{k,j} \prod_{i=1}^n p_t(x_i) \quad (9)$$

Clearly Eq. 8 and Eq. 9 share the same functional form and the only difference lies in that the edge weight used in Eq. 8 is given by  $\frac{\alpha_{k,j}}{\sum_{j'} \alpha_{k,j'}}$  while the edge weight used in Eq. 9 is given by  $w_{k,j}$ , both of which are constrained to be positive and locally normalized. This implies that in order to compute the value of Eq. 8, we can replace all the edge weights  $w_{k,j}$  with  $\frac{\alpha_{k,j}}{\sum_{j'} \alpha_{k,j'}}$  in  $\mathcal{S}$ , and a bottom-up pass evaluation of  $\mathcal{S}$  will give us the desired result at the root of the network. In other words, we reduce the original problem to a joint inference problem in  $\mathcal{S}$  with a set of weights determined by  $\alpha$ . The linear time and space complexity then follows from the linear time and space inference complexity of SPNs.  $\blacksquare$

To evaluate Eq. 7, we also need to compute  $\sum_{\mathcal{T}_t \in \mathcal{T}_T} c_t u_t$  efficiently. Since  $\sum_{\mathcal{T}_t \in \mathcal{T}_T} c_t u_t$  contains a subset of monomials that contain  $w_{k,j}$  in  $Z_{\mathbf{x}}$ , a brute force computation is infeasible in the worst case. The key observation is that we can use the *differential trick* to solve this problem by realizing the fact that  $Z_{\mathbf{x}}$  is a multilinear function in  $\frac{\alpha_{k,j}}{\sum_{j'} \alpha_{k,j'}}$ ,  $\forall k, j$  and has a compact factorization.

**Lemma 3.2.**  $\sum_{\mathcal{T}_t \in \mathcal{T}_T} c_t u_t$  and  $\sum_{\mathcal{T}_t \in \mathcal{T}_F} c_t u_t$  can be computed in  $O(|\mathcal{S}|)$  time and space in a top-down differentiation pass of  $\mathcal{S}$ .

*Proof.* Define  $w_{k,j} \triangleq \frac{\alpha_{k,j}}{\sum_{j'} \alpha_{k,j'}}$ , then

$$\begin{aligned} \sum_{\mathcal{T}_t \in \mathcal{T}_T} c_t u_t &= \sum_{\mathcal{T}_t \in \mathcal{T}_T} \prod_{(k',j') \in \mathcal{T}_{tE}} w_{k',j'} \prod_{i=1}^n p_t(x_i) \\ &= w_{k,j} \sum_{\mathcal{T}_t \in \mathcal{T}_T} \prod_{\substack{(k',j') \in \mathcal{T}_{tE} \\ (k',j') \neq (k,j)}} w_{k',j'} \prod_{i=1}^n p_t(x_i) + 0 \cdot \sum_{\mathcal{T}_t \in \mathcal{T}_F} c_t u_t \\ &= w_{k,j} \left( \frac{\partial}{\partial w_{k,j}} \sum_{\mathcal{T}_t \in \mathcal{T}_T} \prod_{(k',j') \in \mathcal{T}_{tE}} w_{k',j'} \prod_{i=1}^n p_t(x_i) + \frac{\partial}{\partial w_{k,j}} \sum_{\mathcal{T}_t \in \mathcal{T}_F} c_t u_t \right) \\ &= w_{k,j} \left( \frac{\partial}{\partial w_{k,j}} \sum_{\mathcal{T}_t \in \mathcal{T}_T} c_t u_t + \frac{\partial}{\partial w_{k,j}} \sum_{\mathcal{T}_t \in \mathcal{T}_F} c_t u_t \right) \\ &= w_{k,j} \left( \frac{\partial}{\partial w_{k,j}} \sum_{t=1}^{\tau_S} c_t u_t \right) \end{aligned}$$

where the second equality is by Corollary 2.1 that the network polynomial is a multilinear function of  $w_{k,j}$  and the third equality holds because  $\mathcal{T}_F$  is the set of trees that do not contain  $w_{k,j}$ . The rest of the equalities follow by simple algebraic transformations. In summary, the above lemma holds because of the fact that differential operator applied to a multilinear function acts as a selector for all the monomials containing a specific variable, as illustrated in Example 1.1. Hence,

$$\sum_{\mathcal{T}_t \in \mathcal{T}_F} c_t u_t = \sum_{t=1}^{\tau_S} c_t u_t - \sum_{\mathcal{T}_t \in \mathcal{T}_T} c_t u_t$$

can also be computed. To show the linear time and space complexity, recall that the differentiation w.r.t.  $w_{k,j}$  can be efficiently computed by back-propagation in a top-down pass of  $\mathcal{S}$  once we have computed  $\sum_{t=1}^{\tau_S} c_t u_t$  in a bottom-up pass of  $\mathcal{S}$ .  $\blacksquare$

Define  $D_k(\mathbf{x} \mid \mathbf{w}) = \partial V_{\text{root}}(\mathbf{x} \mid \mathbf{w}) / \partial V_k(\mathbf{x} \mid \mathbf{w})$ . Then the differentiation term  $\frac{\partial \sum_{t=1}^{\tau_S} c_t u_t}{\partial w_{k,j}}$  can be computed via back-propagation in a top-down pass of the network as follows:

$$\frac{\partial \sum_{t=1}^{\tau_S} c_t u_t}{\partial w_{k,j}} = \frac{\partial V_{\text{root}}(\mathbf{x} \mid \mathbf{w})}{\partial V_k(\mathbf{x} \mid \mathbf{w})} \frac{\partial V_k(\mathbf{x} \mid \mathbf{w})}{\partial w_{k,j}} = D_k(\mathbf{x} \mid \mathbf{w}) V_j(\mathbf{x} \mid \mathbf{w}) \quad (10)$$

**Theorem 3.1.** For each edge  $(k, j)$ , Eq. 7 can be computed in  $O(|S|)$  time and space.

*Proof.* The proof simply follows from Lemma 3.1 and Lemma 3.2 with the assumption that the moments under the prior has closed form solution.  $\blacksquare$

Let  $\lambda_{k,j} = (w_{k,j} V_j(\mathbf{x} \mid \mathbf{w}) D_k(\mathbf{x} \mid \mathbf{w})) / V_{\text{root}}(\mathbf{x} \mid \mathbf{w})$  and  $f_{k,j} = f(w_{k,j})$ . The final formula for the exact moment of  $w_{k,j}$  under the one-step update posterior  $p$  is given by

$$M_p(f_{k,j}) = (1 - \lambda_{k,j}) M_{p_0}(f_{k,j}) + \lambda_{k,j} M_{p'_0}(f_{k,j}) \quad (11)$$

**Remark 1.** Recall that  $\forall k, j$ , we have  $0 \leq \lambda_{k,j} \leq 1$ :

$$\lambda_{k,j} = \frac{1}{Z_{\mathbf{x}}} \sum_{\mathcal{T}_t \in \mathcal{T}_T} c_t u_t = \frac{1}{p(\mathbf{x} \mid \mathbf{w})} \sum_{\mathcal{T}_t \in \mathcal{T}_T} \prod_{(k,j) \in \mathcal{T}_{tE}} w_{k,j} \prod_{i=1}^n p_t(x_i) \quad (12)$$

where we remind the readers that  $\mathcal{T}_T$  is the set of induced trees that contain the edge  $(k, j)$ . Essentially, Eq. 12 means that  $\lambda_{k,j}$  is the ratio of all the induced trees that contain edge  $(k, j)$  to the network. Roughly speaking, this measures how important the contribution of a specific edge is to the whole network polynomial. As a result, we can interpret Eq. 11 as follows: the more important the edge is, the more portion of the moment comes from the new observation.

**Remark 2.** CCCP for SPNs was originally derived using a sequential convex relaxation technique, where in each iteration a concave surrogate function is constructed and optimized. The key update in each iteration of CCCP (Zhao et al. (2016b) Eq. (7)) is given as follows:

$$w'_{k,j} \propto w_{k,j} \frac{V_j(\mathbf{x} \mid \mathbf{w})}{V_{\text{root}}(\mathbf{x} \mid \mathbf{w})} D_k(\mathbf{x} \mid \mathbf{w}) \quad (13)$$

Note that the R.H.S. of Eq. 13 is exactly the same as  $\lambda_{k,j}$  defined in Eq. 11. From this perspective, CCCP can also be understood as implicitly applying the differential trick to compute  $\lambda_{k,j}$ , i.e., the relative importance of edge  $(k, j)$ , and then take updates according to this importance measure.

**Remark 3.** Note that  $\lambda_{k,j}$  only depends on three terms, i.e., the forward evaluation value  $V_j$ , the backward differentiation value  $D_k$  and the original weight of the edge  $w_{k,j}$ . Readers familiar with backpropagation in neural networks may find this functional form to be similar to the *delta rule* (Hecht-Nielsen et al., 1988), where the weight update for each neuron depends on the forward input value, the backward error signal as well as the strength of the connection between two neurons. This is quite interesting by itself as it makes a more close connection of SPNs to traditional neural networks. In fact, one can indeed interpret SPNs as a special kind of feed-forward neural networks with two kinds of activation functions, and sparse connections between neurons to satisfy the structural constraints imposed by the completeness and decomposability.

Finally, we summarize the linear time algorithm for moment computation in Alg. 1.

**Algorithm 1** Linear Time Exact Moment Computation**Input:** Prior  $p_0(\mathbf{w} \mid \boldsymbol{\alpha})$ , moment  $f$ , SPN  $\mathcal{S}$  and input  $\mathbf{x}$ .**Output:**  $M_p(f(w_{k,j})), \forall (k, j)$ .

- 1:  $w_{k,j} \leftarrow \frac{\alpha_{k,j}}{\sum_{j'} \alpha_{k,j'}}, \forall (k, j)$ .
- 2: Compute  $M_{p_0}(f(w_{k,j}))$  and  $M_{p'_0}(f(w_{k,j})), \forall (k, j)$ .
- 3: Bottom-up evaluation pass of  $\mathcal{S}$  with input  $\mathbf{x}$ . Record  $V_k(\mathbf{x} \mid \mathbf{w})$  at each node  $k$ .
- 4: Top-down differentiation pass of  $\mathcal{S}$  with input  $\mathbf{x}$ . Record  $D_k(\mathbf{x} \mid \mathbf{w})$  at each node  $k$ .
- 5: Compute the exact moment for each  $(k, j)$  based on Eq. 11.

**Corollary 3.1.** Alg. 1 runs in  $O(|\mathcal{S}|)$  time and space.

*Proof.* To see this, from Eq. 10, we only need to cache the forward evaluation value  $V_j$  and the backward differentiation value  $D_k$  for each edge  $(k, j)$ . This can be achieved by two additional copies of the network. Note that line 2 of Alg. 1 can be computed in  $O(|\mathcal{S}|)$  for all the edges. Line 3-5 can all be computed in  $O(|\mathcal{S}|)$ , so the corollary follows.  $\blacksquare$

**4. Online Moment Matching**

In this section we use Alg. 1 as a sub-routine to develop a new Bayesian online learning algorithm for SPNs based on assumed density filtering (Sorenson and Stubberud, 1968). To do so, we find an approximate distribution by minimizing the KL divergence between the one-step update posterior and the approximate distribution. Let  $\mathcal{P} = \{q \mid q = \prod_{k=1}^M \text{Dir}(w_k; \beta_k)\}$ , i.e.,  $\mathcal{P}$  is the space of product of Dirichlet densities that are decomposable over all the sum nodes in  $\mathcal{S}$ . Note that since  $p_0(\mathbf{w}; \boldsymbol{\alpha})$  is fully decomposable, we have  $p_0 \in \mathcal{P}$ . One natural choice is to try to find an approximate distribution  $q \in \mathcal{P}$  such that  $q$  minimizes the KL-divergence between  $p(\mathbf{w} \mid \mathbf{x})$  and  $q$ , i.e.,

$$\hat{p} = \arg \min_{q \in \mathcal{P}} \mathbb{KL}(p(\mathbf{w} \mid \mathbf{x}) \parallel q) \quad (14)$$

It is not hard to show (proof in appendix) that when  $q$  is an exponential family distribution, which is the case in our setting, the above minimization problem corresponds to solving the following moment matching equation:

$$\mathbb{E}_{p(\mathbf{w} \mid \mathbf{x})}[T(w_k)] = \mathbb{E}_{q(w_k)}[T(w_k)] \quad (15)$$

where  $T(w_k)$  is the vector of sufficient statistics of  $q(w_k)$ . When  $q(\cdot)$  is a Dirichlet, we have  $T(w_k) = \log w_k$ . This principle of finding an approximate distribution is also known as *reverse information projection* in the literature of information theory (Csiszár and Matus, 2003).<sup>2</sup> By utilizing our efficient linear time algorithm for exact moment computation, we propose a Bayesian online learning algorithm for SPNs based on the above moment matching principle, called ADF. The pseudocode is shown in Alg. 2. Details on how to solve the moment matching equation (Eq. 15) are presented in appendix. As a note, we can also extend the above ADF algorithm to the batch setting via a recently proposed technique known as stochastic expectation propagation (Li et al., 2015).

2. As a comparison, information projection corresponds to minimizing  $\mathbb{KL}(q \parallel p(\mathbf{w} \mid \mathbf{x}))$  within the same family of distributions  $q \in \mathcal{P}$ . Finding an approximate distribution for SPNs based on information projection has recently been studied by Zhao et al. (2016a), and an algorithm called CVB-SPN is proposed therein.

**Algorithm 2** Assumed Density Filtering for SPN**Input:** Prior  $p_0(\mathbf{w} \mid \boldsymbol{\alpha})$ , SPN  $\mathcal{S}$  and input  $\{\mathbf{x}_i\}_{i=1}^{\infty}$ .

- 1:  $p(\mathbf{w}) \leftarrow p_0(\mathbf{w} \mid \boldsymbol{\alpha})$
- 2: **for**  $i = 1, \dots, \infty$  **do**
- 3:   Apply Alg. 1 to compute  $\mathbb{E}_{p(\mathbf{w} \mid \mathbf{x}_i)}[\log w_{k,j}]$  for all edges  $(k, j)$ .
- 4:   Find  $\hat{p} = \arg \min_{q \in \mathcal{P}} \mathbb{KL}(p(\mathbf{w} \mid \mathbf{x}_i) \parallel q)$  by solving the moment matching equation Eq. 15.
- 5:    $p(\mathbf{w}) \leftarrow \hat{p}(\mathbf{w})$ .
- 6: **end for**

## 5. Experiments

### 5.1 Experimental Setting

We conduct experiments on 20 real-world data sets that have been used as benchmarks to evaluate the effectiveness of learning algorithms for SPNs (Gens and Domingos, 2013; Rooshenas and Lowd, 2014; Zhao et al., 2016a; Adel et al., 2015; Vergari et al., 2015). Statistics about the 20 data sets and their corresponding SPN models are shown in Table 1. All the random variables in these 20 data sets are binary. We use LearnSPN as the structure learning algorithm to build structures for the 20 data sets. Since LearnSPN, along with other structure learning algorithms for SPNs (Adel et al., 2015; Rooshenas and Lowd, 2014), will only return tree structured networks, we post-process the constructed networks by merging leaf nodes with same distributions, leading to general networks with DAG structures. Note that more complicated and compression-efficient post-processing strategy exists for SPNs (Rahman and Gogate, 2016). We emphasize how we are not interested in obtaining the most compact SPN representations. Instead, we would like to investigate how the proposed linear time moment computation algorithm scales on general SPNs with DAG structures, compared with the existing quadratic moment computation algorithm. As the last step, we equivalently transform the network structures by removing consecutive sum nodes or product nodes, using the same technique introduced by Vergari et al. (2015). This step helps to reduce the size of the networks while keeping the distributions unchanged. As a result, the size of the reduced network is roughly twice the number of parameters to be estimated, as shown in Table 1.

In the experiments we aim to show that using the proposed linear time algorithm for computing desired moments of the one-step update posterior, both BMM and ADF can run in linear time, having the same order of time complexity as the other online learning algorithms for SPNs. ADF and BMM are methods based on the general principle of moment matching, but they differ at which moments to match. On the other hand, we would also like to have a fair and extensive empirical comparison among all the online parameter learning algorithms for SPNs proposed to date, including projected gradient descent (PGD) (Poon and Domingos, 2011; Gens and Domingos, 2012), exponentiated gradient (EG) (Kivinen and Warmuth, 1997), sequential monomial approximation (SMA) (Zhao et al., 2016b), concave-convex procedure/expectation maximization (CCCP) (Peharz, 2015; Zhao et al., 2016b), collapsed variational Bayes (CVB) (Zhao et al., 2016a), online Bayesian moment matching (BMM) (Rashwan et al., 2016) and assumed density filtering (ADF). We compare both the training time and the test set log-likelihood scores of those online learning algorithms, evaluating both the computational efficiency as well as the statistical performance. We implement all the above learning algorithms in C++ and conduct our experiments on a server with 32 E5-2650 2.00GHz CPUs. For

Bayesian online learning algorithms, we use their posterior means as the estimators to evaluate on test data sets, and we use uniform Dirichlets as priors.

## 5.2 Results

All the SPNs used in our experiments are DAGs, by using the merging techniques introduced in the last section. As a result, the moment computation algorithm introduced in Rashwan et al. (2016) for trees cannot be applied here. A naive algorithm for moment computation on DAGs scales quadratically in the size of the network, and it does not terminate within 24 hours even for the smallest SPN (NLTCs). As a comparison, by using Alg. 1 to compute the exact moments of the posterior distribution after each observation, both BMM and ADF can be run in linear time in the size of the network as well as the size of the training data set. Alg. 1 significantly broadens the applicability of the method of moments to the set of SPNs used in practice. We plot the runtime of different online learning algorithms on the 20 data sets, and sort them according to the time used. From Fig. 1 it is clear that the runtime of these 7 algorithms are within the same order. This is as expected since all the 7 algorithms have linear time complexity. In practice BMM and ADF are faster than all the other methods except CCCP. For example, on the netflix data set, both ADF and BMM uses around 55 hours while SMA uses around 84 hours. All the experiments are restricted to run in 84 hours. None of the 7 online algorithms finishes within 84 hours on the 20Newsdp data set.

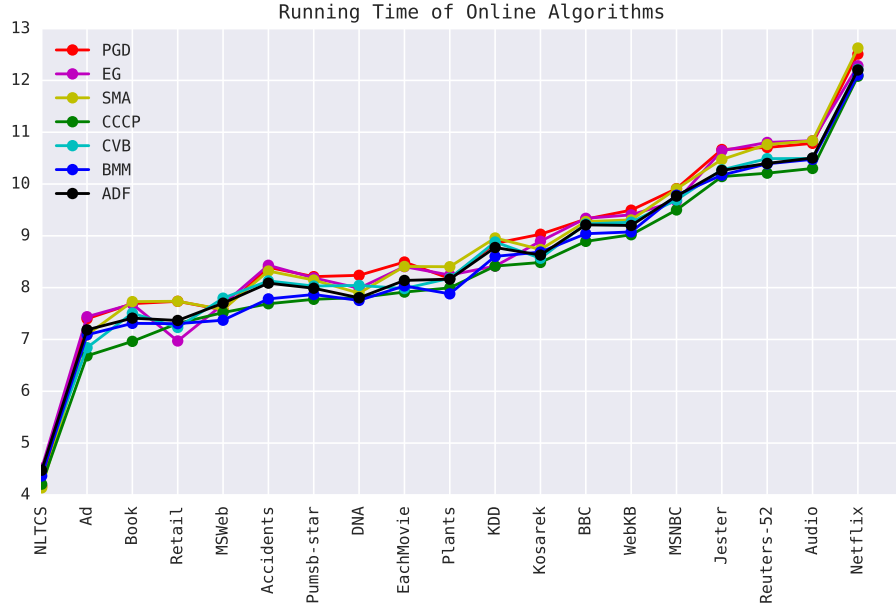


Figure 1: Runtime (log seconds) of online learning algorithms for SPNs on 20 data sets. BMM and ADF are faster than all the other algorithms except CCCP.

We summarize the test set log-likelihood scores for the 7 algorithms in Table 2. Among all the algorithms, online CCCP achieves the best performance on 15 out of 20 data sets. Online CVB is second to CCCP, obtaining 4 highest test set scores. BMM and ADF are comparable to each other, while both of these two algorithms perform slightly worse than SMA. Except PGD, all the other algorithms admit multiplicative updates. Based on our experiments, EG is the one that is most

sensitive to the choice of learning rates, while BMM and ADF are free of learning rate. The fact that both ADF and BMM usually perform worse than CCCP and SMA can be understood from their design principles: while CCCP and SMA are optimization based algorithms that are explicitly designed to maximize the log-likelihood scores on the training set, BMM and ADF are projection based algorithms by moment matching. By the concentration of the log-likelihood scores, one may also expect CCCP and SMA to have good log-likelihood scores on the test set provided these two algorithms achieve good scores on the training data set. On the other hand, both ADF and BMM work by matching specific moments, and this usually provides a tradeoff between computational efficiency and statistical performance. As a take-home-message, we recommend CCCP (if one does not have any prior information about the weights) and CVB (if one has useful prior information about the weights) if one is more interested in building good generative models, or BMM/ADF if one cares more about the speed of learning.

## 6. Conclusion

We propose a linear time algorithm to efficiently compute the moments of model parameters in SPNs. The key technique used in the design of our algorithm is the differential trick that is able to efficiently evaluate a multilinear function given that the network polynomial admits a tractable factorization. Using the proposed algorithm as a sub-routine, we are able to improve the time complexity of BMM from quadratic to linear on general SPNs with DAG structures. We also use this sub-routine to design a new online algorithm, ADF. Extensive experiments are conducted to evaluate the online learning algorithms proposed to date. We show that by using the proposed moment computation algorithm ADF and BMM are faster than all the competitors except CCCP. As a future direction, we hope to apply the proposed moment computation algorithm in the design of efficient structure learning algorithms for SPNs. We also expect that the same differential trick that we make explicit here might find other uses for SPN learning in the future.

## References

- T. Adel, D. Balduzzi, and A. Ghodsi. Learning the structure of sum-product networks via an svd-based algorithm. In *Proceedings of UAI*, 2015.
- C. Boutilier, N. Friedman, M. Goldszmidt, and D. Koller. Context-specific independence in Bayesian networks. In *Proceedings of the Twelfth international conference on Uncertainty in artificial intelligence*, pages 115–123. Morgan Kaufmann Publishers Inc., 1996.
- I. Csiszár and F. Matus. Information projections revisited. *IEEE Transactions on Information Theory*, 49(6):1474–1490, 2003.
- A. Darwiche. A differential approach to inference in bayesian networks. *Journal of the ACM (JACM)*, 50(3):280–305, 2003.
- A. Dennis and D. Ventura. Greedy structure search for sum-product networks. In *International Joint Conference on Artificial Intelligence*, volume 24, 2015.
- R. Gens and P. Domingos. Discriminative learning of sum-product networks. In *Advances in Neural Information Processing Systems*, pages 3248–3256, 2012.

- R. Gens and P. Domingos. Learning the structure of sum-product networks. In *Proceedings of The 30th International Conference on Machine Learning*, pages 873–880, 2013.
- R. Hecht-Nielsen et al. Theory of the backpropagation neural network. *Neural Networks*, 1 (Supplement-1):445–448, 1988.
- P. Jaini, A. Rashwan, H. Zhao, Y. Liu, E. Banijamali, Z. Chen, and P. Poupart. Online algorithms for sum-product networks with continuous variables. In *Proceedings of the Eighth International Conference on Probabilistic Graphical Models*, pages 228–239, 2016.
- J. Kivinen and M. K. Warmuth. Exponentiated gradient versus gradient descent for linear predictors. *Information and Computation*, 132(1):1–63, 1997.
- Y. Li, J. M. Hernández-Lobato, and R. E. Turner. Stochastic expectation propagation. In *Advances in Neural Information Processing Systems*, pages 2323–2331, 2015.
- J. D. Park and A. Darwiche. A differential semantics for jointree algorithms. *Artificial Intelligence*, 156(2):197–216, 2004.
- R. Peharz. *Foundations of Sum-Product Networks for Probabilistic Modeling*. PhD thesis, Graz University of Technology, 2015.
- R. Peharz, S. Tschiatschek, F. Pernkopf, and P. Domingos. On theoretical properties of sum-product networks. In *AISTATS*, 2015.
- R. Peharz, R. Gens, F. Pernkopf, and P. Domingos. On the latent variable interpretation in sum-product networks. *arXiv preprint arXiv:1601.06180*, 2016.
- H. Poon and P. Domingos. Sum-product networks: A new deep architecture. In *Proc. 12th Conf. on Uncertainty in Artificial Intelligence*, pages 2551–2558, 2011.
- T. Rahman and V. Gogate. Merging strategies for sum-product networks: From trees to graphs. In *Proceedings of the Thirty-Second Conference on Uncertainty in Artificial Intelligence, UAI*, 2016.
- A. Rashwan, H. Zhao, and P. Poupart. Online and distributed bayesian moment matching for parameter learning in sum-product networks. In *Proceedings of the 19th International Conference on Artificial Intelligence and Statistics*, pages 1469–1477, 2016.
- A. Rooshenas and D. Lowd. Learning sum-product networks with direct and indirect variable interactions. In *ICML*, 2014.
- H. W. Sorenson and A. R. Stubberud. Non-linear filtering by approximation of the a posteriori density. *International Journal of Control*, 8(1):33–51, 1968.
- A. Vergari, N. Di Mauro, and F. Esposito. Simplifying, regularizing and strengthening sum-product network structure learning. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, pages 343–358. Springer, 2015.
- H. Zhao, M. Melibari, and P. Poupart. On the Relationship between Sum-Product Networks and Bayesian Networks. In *ICML*, 2015.



H. Zhao, T. Adel, G. Gordon, and B. Amos. Collapsed variational inference for sum-product networks. In *ICML*, 2016a.

H. Zhao, P. Poupart, and G. Gordon. A unified approach for learning the parameters of sum-product networks. *NIPS*, 2016b.

## Appendix A. Details about Solving Moment Matching Equations in BMM and ADF

In the appendix we show that minimizing the KL divergence between the one-step update posterior distribution and the approximate distribution leads to a moment matching equation to be solved when the approximate  $q$  comes from the exponential family distributions. We will also show in detail the system of equations to be solved in both BMM and ADF for SPNs.

Starting from the approximate distribution  $q = q(\mathbf{w}; \theta)$  with natural parameter  $\theta$ , since  $q$  is assumed to be an exponential family distribution, we have

$$q(\mathbf{w}; \theta) = h(\mathbf{w})g(\theta) \exp(\theta^T T(\mathbf{w}))$$

where  $T(\mathbf{w})$  is the sufficient statistics of the exponential family distribution. Consider the minimization problem over  $q$ , we have

$$\begin{aligned} \min_{q \in \mathcal{P}} \quad \mathbb{KL}(p(\mathbf{w} | \mathbf{x}) || q) &= \min_{q \in \mathcal{P}} \int_{\mathbf{w}} p(\mathbf{w} | \mathbf{x}) \log \frac{p(\mathbf{w} | \mathbf{x})}{q(\mathbf{w}; \theta)} d\mathbf{w} \\ &\Leftrightarrow \max_{\theta} \int_{\mathbf{w}} p(\mathbf{w} | \mathbf{x}) \log h(\mathbf{w})g(\theta) \exp(\theta^T T(\mathbf{w})) d\mathbf{w} \\ &\Leftrightarrow \max_{\theta} \int_{\mathbf{w}} p(\mathbf{w} | \mathbf{x}) (\log g(\theta) + \theta^T T(\mathbf{w})) d\mathbf{w} \end{aligned}$$

Note that  $g(\theta)$ , the partition function, of  $q$  is guaranteed to be log-concave since  $q$  is assumed to be an exponential family distribution. It follows that the above maximization problem is a smooth concave maximization problem where the optimal  $\theta^*$  must have  $\mathbf{0}$  gradient. Differentiating the above function w.r.t.  $\theta$ , leading to the following equation to be solved:

$$-\nabla_{\theta} \log g(\theta) = \mathbb{E}_{p(\mathbf{w}|\mathbf{x})}[T(\mathbf{w})]$$

Realizing  $-\nabla_{\theta} \log g(\theta) = \mathbb{E}_{q(\mathbf{w})}[T(\mathbf{w})]$ , we arrive at the moment matching Eq. 15:

$$\mathbb{E}_{p(\mathbf{w}|\mathbf{x})}[T(w_k)] = \mathbb{E}_{q(\mathbf{w})}[T(w_k)]$$

For  $q$  being a product of Dirichlets, we have  $T(w_k) = \log w_k$ , where the log is understood to be taken elementwisely. In the ADF algorithm, for each edge  $w_{k,j}$  the above moment matching equation amounts to solving the following equation:

$$\psi(\theta_{k,j}) - \psi\left(\sum_{j'} \theta_{w,j'}\right) = \mathbb{E}_{p(\mathbf{w}|\mathbf{x})}[\log w_{k,j}]$$

where  $\psi(\cdot)$  is the digamma function. This is a system of nonlinear equation about  $\theta$  where the R.H.S. of the above equation can be computed using Alg. 1 in  $O(|S|)$  time for all the edges  $(k, j)$ .

To efficiently solve it, we take  $\exp(\cdot)$  at both sides of the equation and approximate the L.H.S. using the fact that  $\exp(\psi(\theta_{k,j})) \approx \theta_{k,j} - \frac{1}{2}$  for  $\theta_{k,j} > 1$ . Expanding the R.H.S. of the above equation using the identity from Eq. 11, we have:

$$\begin{aligned} \exp\left(\psi(\theta_{k,j}) - \psi\left(\sum_{j'} \theta_{w,j'}\right)\right) &= \exp\left(\mathbb{E}_{p(\mathbf{w}|\mathbf{x})}[\log w_{k,j}]\right) \\ \Leftrightarrow \frac{\theta_{k,j} - \frac{1}{2}}{\sum_{j'} \theta_{k,j'} - \frac{1}{2}} &= \left(\frac{\alpha_{k,j} - \frac{1}{2}}{\sum_{j'} \alpha_{k,j'} - \frac{1}{2}}\right)^{(1-\lambda_{k,j})} \times \left(\frac{\alpha_{k,j} + \frac{1}{2}}{\sum_{j'} \alpha_{k,j'} + \frac{1}{2}}\right)^{\lambda_{k,j}} \end{aligned} \quad (16)$$

Note that  $\frac{\alpha_{k,j} - \frac{1}{2}}{\sum_{j'} \alpha_{k,j'} - \frac{1}{2}}$  is approximately the mean of the prior Dirichlet under  $p_0$  and  $\frac{\alpha_{k,j} + \frac{1}{2}}{\sum_{j'} \alpha_{k,j'} + \frac{1}{2}}$  is approximately the mean of  $p'_0$ , where  $p'_0$  is the posterior by adding one pseudo-count to  $w_{k,j}$ . So Eq.16 is essentially finding a posterior with hyperparameter  $\theta$  such that the posterior mean is approximately the weighted geometric mean of the means given by  $p_0$  and  $p'_0$ , weighting by  $\lambda_{k,j}$ . From line 1 of Alg. 1, since the only thing we need is the mean of the prior at each iteration, we do not need to fully solve the above equation for  $\theta$ . What we need is the normalized  $\theta$ , i.e., the mean of the prior Dirichlet at the beginning of each iteration. This observation helps us to get rid of solving the above equation exactly. We can simply use the R.H.S. of Eq. 16 to approximate the mean of the prior at next iteration. In practice this approximation greatly accelerates ADF and as long as  $\theta_{k,j} > 1$ , the approximation is accurate.

Instead of matching the moments given by the sufficient statistics, also known as the natural moments, BMM is trying to find an approximate distribution  $q$  by matching the first order moments, i.e., the mean of the prior and the one-step update posterior. Using the same notation, we want  $q$  to match the following equation:

$$\mathbb{E}_{q(\mathbf{w})}[w_k] = \mathbb{E}_{p(\mathbf{w}|\mathbf{x})}[w_k] \quad \Leftrightarrow \quad \frac{\theta_{k,j}}{\sum_{j'} \theta_{k,j'}} = (1 - \lambda_{k,j}) \frac{\alpha_{k,j}}{\sum_{j'} \alpha_{k,j'}} + \lambda_{k,j} \frac{\alpha_{k,j} + 1}{\sum_{j'} \alpha_{k,j'} + 1} \quad (17)$$

Again, we can interpret the above equation as to finding the posterior hyperparameter  $\theta$  such that the posterior mean is given by the weighted arithmetic mean of the means given by  $p_0$  and  $p'_0$ , weighting by  $\lambda_{k,j}$ . Notice that due to the normalization constraint, we cannot solve for  $\theta$  directly from the above equations, and in order to solve for  $\theta$  we will need one more equation to be added into the system. However, from line 1 of Alg. 1 again, what we really need is not  $\theta$ , but only its normalized version in the next iteration. So we can get rid of the additional equation and use Eq. 17 as the update formula directly in our algorithm.

This completes the discussion about implementation details of Alg. 2 and BMM for SPNs.

Table 1: Statistics of the experiments.  $n$  is the number of random variables/features in the data set,  $|S|$  is the size of the network, including nodes and edges.  $p$  is the number of parameters to be estimated.  $r$  means the ratio of training instances times the number of variables to the number of parameters.

Data set	$n$	$ S $	$p$	Train	Valid	Test	$r$
NLTCS	16	3,515	1,716	16,181	2,157	3,236	150.87
MSNBC	17	49,367	24,452	291,326	38,843	58,265	202.54
KDD	64	28,833	14,292	180,092	19,907	34,955	806.46
Plants	69	118,251	58,853	17,412	2,321	3,482	20.41
Audio	100	1,479,413	739,525	15,000	2,000	3,000	2.03
Jester	100	1,964,885	982,310	9,000	1,000	4,116	0.92
Netflix	100	5,904,799	2,952,297	15,000	2,000	3,000	0.51
Accidents	111	151,539	74,804	12,758	1,700	2,551	18.93
Retail	135	42,439	21,044	22,041	2,938	4,408	141.40
Pumsb-star	163	128,847	63,173	12,262	1,635	2,452	31.64
DNA	180	782,291	390,907	1,600	400	1,186	0.74
Kosarek	190	107,119	53,204	33,375	4,450	6,675	119.19
MSWeb	294	41,589	20,346	29,441	3,270	5,000	425.42
Book	500	83,531	41,122	8,700	1,159	1,739	105.78
EachMovie	500	378,015	188,387	4,524	1,002	591	12.01
WebKB	839	1,761,941	879,893	2,803	558	838	2.67
Reuters-52	889	2,509,457	1,253,390	6,532	1,028	1,540	4.63
20Newsgp	910	16,593,181	8,295,407	11,293	3,764	3,764	1.24
BBC	1058	2,027,427	1,012,604	1,670	225	330	1.74
Ad	1556	169,773	83,103	2,461	327	491	46.08

Table 2: Average log-likelihoods on test data. Highest scores are highlighted in bold.

Data set	PGD	EG	SMA	CCCP	CVB	BMM	ADF
NLTCS	-8.74	-6.69	-6.52	<b>-6.09</b>	-6.21	-7.88	-7.15
MSNBC	-8.08	-6.83	-6.90	<b>-6.23</b>	-6.38	-7.63	-7.51
KDD	-27.82	-2.36	-2.35	<b>-2.13</b>	-2.15	-4.84	-6.37
Plants	-29.42	-22.35	-20.36	<b>-13.80</b>	-16.18	-23.25	-23.96
Audio	-53.06	-44.87	-50.22	<b>-40.16</b>	-41.49	-50.18	-53.76
Jester	-63.48	-57.76	-63.02	<b>-53.05</b>	-54.29	-60.23	-62.82
Netflix	-67.09	-64.72	-67.50	-57.99	<b>-57.96</b>	-63.78	-65.28
Accidents	-47.20	-50.44	-42.36	<b>-36.67</b>	-38.76	-47.81	-49.97
Retail	-81.88	-12.02	-12.86	<b>-11.09</b>	-11.40	-18.14	-22.53
Pumsb-star	-75.30	-72.34	-56.31	<b>-33.69</b>	-40.55	-79.39	-61.71
DNA	-105.80	-100.87	-106.67	<b>-98.32</b>	-100.42	-106.00	-113.61
Kosarek	-138.50	-14.12	-15.89	<b>-11.24</b>	-11.39	-20.49	-28.55
MSWeb	-130.75	-20.25	-12.42	<b>-11.00</b>	-11.21	-19.76	-29.38
Book	-297.14	-40.70	-41.67	-36.54	<b>-36.22</b>	-68.45	-69.59
EachMovie	-251.95	-79.73	-81.27	<b>-58.03</b>	-65.90	-92.69	-109.53
WebKB	-426.02	-209.92	-180.01	<b>-166.82</b>	-172.21	-224.48	-234.79
Reuters-52	-567.02	-325.40	-113.61	<b>-94.70</b>	-100.62	-162.23	-172.96
20Newsgp	n/a	n/a	n/a	n/a	n/a	n/a	n/a
BBC	-401.47	-280.73	-280.28	-275.94	<b>-275.74</b>	-351.54	-355.97
Ad	-636.44	-118.65	-76.79	-57.96	<b>-56.71</b>	-112.22	-154.89